DL-PA Solver

Victor Mercklé
Supervised by
Andreas Herzig[1], Abdallah Saffidine[2], and Frédéric Maris[2]

[1]IRIT
[2]Univ. of New South Wales UNSW

Wednesday 1st September, 2021

Abstract

We design an algorithm to solve the model-checking problem of Dynamic Logic of Propositional Assignment (DL-PA), which is a variant of dynamic logic. DL-PA is made to encode programs, games and planning tasks. As demonstrated by SAT, generic solvers can be faster and easier to use than algorithms dedicated to the task. DL-PA's model-checking problem is in P-SPACE, the same complexity class as many games that have recently seen great results by using Monte Carlo methods. We describe a game which is equivalent to solving a DL-PA formula, and we implement, optimize and test Monte Carlo methods to produce a DL-PA solver.

Contents

1 Introduction

A specific solver is tailor made for one class of problems. A human expert will create algorithms, data structures and heuristics around the problem to quickly and efficiently solve it. This works in most cases. The downside is that each new problem requires new code and designing or adapting algorithms.

A generic solver for a language will solve all problems that can be expressed in that language. Since one solver will solve many problems, optimizing the solver is worth the time. Though it is usually harder to make a solver fast for every problems. The language chosen dictates which class of problems we can express and solve, and how hard making a fast solver will be.

The concept of using generic solvers for mathematical languages is widely used : linear programming, SAT solving. SAT solvers are the closest to DL-PA and we tried to implement one of the most important ideas in SAT solving, conflict driven clause learning [SS96]. SAT solvers are a success as they are faster in some industrial problems than hand crafted algorithms. However they are the result of years of programming and theory development.

We will develop a solver for a dynamic logic variant: Dynamic Logic of Propositional Assignment (DL-PA). Its model checking problem is in PSPACE-COMPLETE. [BHST14].

DL-PA is well suited to reason about programs, games and planning tasks. It leads to problem descriptions that are more intuitive and natural than say, QBF, which makes the structure of the problem more obvious to the solver. Furthermore, there is currently no solvers available to DL-PA, although it might be possible to translate DL-PA formulae to QBF and use a dedicated QBF solver.

In this report, we investigate whether a DL-PA solver based on game solving methods could be more efficient than the usual algorithms for logic formulae.

First, in section 2 we will introduce the DL-PA language and the Monte Carlo Tree Search (MCTS) algorithm, then we will present how to apply MCTS to the model-checking problem in section 3.

In the section 4 we describe new algorithms based on the MCTS algorithms. It contains modifications to the MCTS algorithm and solvers that do not use Monte Carlo, it also explains in more details the parameters of the MCTS algorithm.

Most of the ideas are implemented and in section 5 we explore the space of the parameters and experimentally find which algorithm is the fastest to solve DL-PA model checking problems. we test the algorithms on the Hanoi puzzle and on more abstract problems. All of the code used and the multicore experiment runner is available at https://github.com/vmerckle/mctsdlpag.

2 Background: DL-PA and MCTS

2.1 DL-PA

Dynamic Logic of Propositional Assignments (DL-PA)[BHT13] is a dynamic logic; it is made to reason about imperative programs. DL-PA extends propositional logic by adding modalities $\langle\pi\rangle$, one per program π. DL-PA's atomic programs are assignments, written $p \leftarrow \psi$ and tests, written $\psi?$. Programs can be composed deterministically and nondeterministically; they also allow for tests and nondeterministic iteration (using the Kleene star *). Those operators capture the assignments and loops used in programming languages.

We now describe in details the DL-PA language with its formal definition, and give one example where we encode a counter in a formula. In the last section, we encode the Hanoï tower puzzle and solve it using our solvers.

We modify DL-PA's grammar by making two changes: first, negation is only allowed for atomic variables and not for formulae; second, $\wedge$, $\vee$ and $\cup$ operators are n-ary operators instead of binary operators. Both modifications are straightforward and reversible thus all the results on DL-PA hold for this grammar. The language of DL-PA in this report is defined by the following grammar:

$$\mathcal{L}_{formulae} : \varphi ::= p \mid \neg p \mid \top \mid \bot \mid \bigvee_i^n \varphi_i \mid \bigwedge_i^n \varphi_i \mid \langle\pi\rangle\varphi \mid [\pi]\varphi$$

$$\mathcal{L}_{programs} : \pi ::= p \leftarrow \varphi \mid \varphi? \mid \pi_1 ; \pi_2 \mid \bigcup_i^n \pi_i \mid \pi^*$$

Additionaly, in this report we will also use an abbreviation for the sequence of $i \in \mathbb{N}$ times the program π: $(\pi;\pi;\ldots\pi)$ i times can be written π^i, with $\pi^0 = \top?$ (does nothing).

Variables $p, a, b, \ldots$ are elements of a countably infinite set of propositional variables denoted $\mathbb{P}$. A valuation $V \in 2^{\mathbb{P}}$ is set of variables. When we assign the value of a variable a to $\top$ in a valuation, it means we add the element a to the valuation. If we assign a to $\bot$, we remove a from the valuation. The interpretation of a DL-PA formula φ is a set of valuations $||\varphi|| \subseteq 2^{\mathbb{P}}$. The interpretation of a program π is a binary relation $||\pi|| \subseteq 2^{\mathbb{P}} \times 2^{\mathbb{P}}$. All the interpretations are given in the following table:

$$
\begin{aligned}
||p|| &= \{V :\ p \in V\}\\
||\neg p|| &= \{V :\ p \notin V\}\\
||\top|| &= 2^{\mathbb{P}}\\
||\bot|| &= \emptyset\\
\left|\left|\bigvee_{i}^{n} \varphi_i\right|\right| &= \bigcup_{i}^{n} ||\varphi_i||\\
\left|\left|\bigwedge_{i}^{n} \varphi_i\right|\right| &= \bigcap_{i}^{n} ||\varphi_i||\\
||\langle\pi\rangle\varphi|| &= \{V :\ \exists W/(V,W) \in ||\pi|| \text{ and } W \in ||\varphi||\}\\
||[\pi]\varphi|| &= \{V :\ \forall W/(V,W) \in ||\pi|| \text{ and } W \in ||\varphi||\}\\
||p \leftarrow \varphi|| &= \{(V,W) :\ \text{either } V \in ||\varphi|| \text{ and } W = V \cup \{p\}, \text{or } V \notin ||\varphi|| \text{ and } W = V \setminus \{p\}\}\\
||\varphi?|| &= \{(V,V) :\ V \in ||\varphi||\}\\
||\pi_1 ; \pi_2|| &= \{(V,W) :\ \exists U/(V,U) \in ||\pi_1|| \text{ and } (U,V) \in ||\pi_2||\}\\
\left|\left|\bigcup_{i}^{n} \pi_i\right|\right| &= \bigcup_{i}^{n} ||\pi_i||\\
||\pi^*|| &= \bigcup_{i \in \mathbb{N}} ||\pi||^i
\end{aligned}
$$

We describe the program operators informally : $p \leftarrow \varphi$ assigns p to φ; the test $\varphi?$ does nothing if φ is true, and fails otherwise; $\pi_1 ; \pi_2$ is the deterministic composition; $\pi_1 \cup \pi_2$ is the nondeterministic composition; π^* is the nondeterministic iteration. A DL-PA formula φ has a finite number of variable written $\mathbb{P}_\varphi$, thus the iteration is finite and the Kleene Star can be removed[BHST14].

The model-checking problem, that asks whether the valuation V is a correct model for a formula φ is written $V \models \varphi$ and is equivalent to $V \in ||\varphi||$. The satisfiability problem asks whether there exists a valuation that satisfies φ. We can rewrite the satisfiability problem as a model-checking problem : $\emptyset \models \langle \underset{p \in \mathbb{P}_\varphi}{;} (p \leftarrow \top \cup p \leftarrow \bot)\rangle\varphi$, this is true if and only if there exist a valuation that verifies φ. Therefore in this report we only talk about the model-checking problem. The two problems are in P-SPACE [BHST14].

One example to understand modalities : $\{\} \models \langle a \leftarrow \top \cup b \leftarrow \top\rangle(a \wedge b)$. Applying the deterministic program $a \leftarrow \top$ to $\{\}$ gives only one valuation : $\{\{a\}\}$, likewise $b \leftarrow \top$ gives $\{\{b\}\}$. The non deterministic union $\cup$ of those two programs however, gives two valuations : $\{\{a\},\{b\}\}$. We have finished evaluating the diamond $\langle\rangle$, we are left with $\{\{a\},\{b\}\} \models a \wedge b$. None of the valuations satisfy ab so the formula is false.

We describe another example of encoding a problem into a formula, we give the encoding for a counter. It counts to n using a unary counter. $Val(i)$ encodes that the counter has value i.

$$\begin{aligned}
\mathrm{CountTo}(K) &= Val(K-1)?\,;Val(K) \leftarrow \top\,;Val(K-1) \leftarrow \bot \\
\mathrm{Count} &= \bigcup_{i=1}^{n} \mathrm{CountTo}(i) \\
\{Val(0)\} &\models \langle \mathrm{Count}^* \rangle Val(n)
\end{aligned}$$

The program CountTo(K) counts from $K-1$ to K; It sets the value of $Val(K)$ to $\top$, and the value of $Val(K-1)$ to $\bot$, but only if $Val(K-1)$ is true. Programs can be thought as binary relations between sets of valuation. In this way, CountTo(K) takes every valuation that contains $Val(K-1)$ to a valuation with $Val(K-1)$ removed and $Val(K)$ added. Every valuation without $Val(K-1)$ is associated with the empty set.

Therefore, if we apply the program CountTo(1) to the set of valuation $\{\{Val(0)\}\}$, we obtain $\{\{Val(1)\}\}$. To build a counter, we have to take the union of all the CountTo(K) programs. Each time we apply the union, we increment our counter by one.

This model-checking problem is true, because after applying n times the program Count, $Val(n)$ is true.

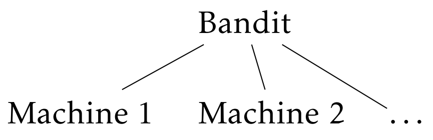
To solve the model-checking problem, we will use methods that are commonly used in games. The next two sections describes parts of the Monte Carlo Tree Search algorithm.

2.2 Definition of the MCTS algorithm

2.2.1 Upper Confidence Bound

We describe the multi-armed problem and one solution : Upper Confidence Bound (UCB [ACF02]). This problem will be encountered in the MCTS algorithm described in the next section.

The multi-armed bandit problem is a classic optimisation problem in which we are faced with multiple slot machines with unknown rewards, and a finite number of tokens for the machines. How do we earn as much money as possible? We need to exploit the machine with the best reward. However, we do not know which one it is. Thus we have to explore all machines and deduce their mean reward.



One way to solve this is the Upper Confidence Bound (UCB) algorithm, which adds together the computed average reward and an *upper confidence bound*. This has the effect of selecting the actions that have the highest average, but also consider those who are promising but uncertain. This deals with the exploration versus exploitation dilemma. C is a real constant, its theoretical optimal value in this scenario is $\sqrt{2}$ but we will see later that in practice the optimal differs.

For each action i that consists of trying the machine i, we associate a number $U(i)$ defined below.

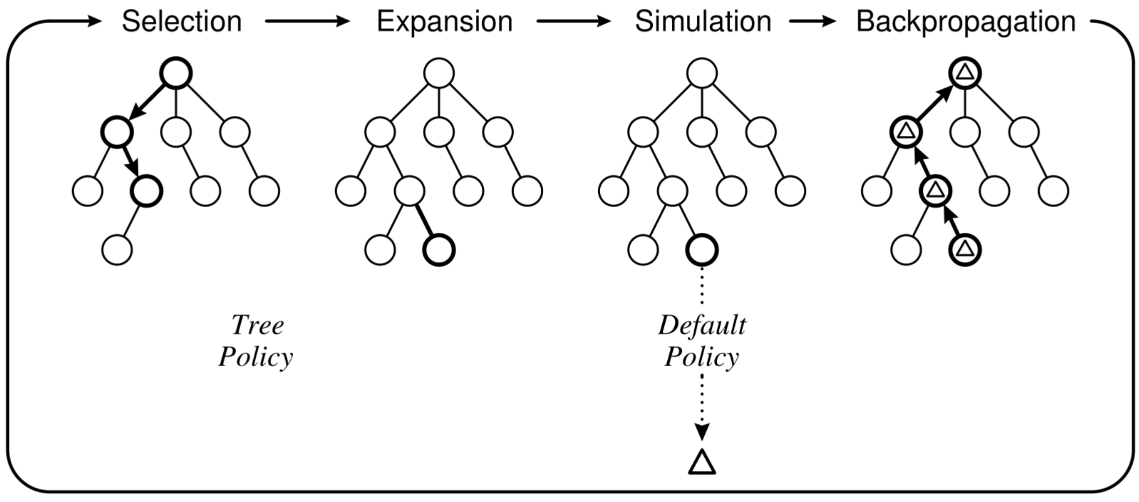


Figure 1: [BPW+12]

$$U(i) = \frac{r_i}{n_i} + C\sqrt{\frac{\ln n}{n_i}}$$

Where r_i is the sum of all the rewards collected by machine i, n_i the number of attempts and n the total number of attempts over all machines. At each iteration, we do the action i which has the highest score $U(i)$ and then we update all the statistics variables r_i, n_i and n.

In the following sections we will describe the Monte Carlo Tree Search algorithm, which can be seen as solving a succession of multi-armed bandit problems.

2.2.2 MCTS

Monte Carlo Tree Search (MCTS)[BPW+12] is an algorithm to quickly find the optimal decision. It is an approach that can be applied directly to domains that can be represented as trees of sequential decisions. Only the evaluation of the final states are required to run this algorithm. MCTS has been very effective in practice in many games such as Go and Chess. It excels against other algorithms in games where we do not have good heuristics to evaluate an intermediate state. This is the case for DL-PA formulae as we will see later.

A node represent a game state that consist of all the data needed to continue the game. The children of a node are the results of an action taken by a player.

The MCTS loop is composed of 4 steps described in Figure 1. At each iteration of the loop, we select the most important node using a tree policy that takes into account the statistics stored in each node. We then add a child to this node that corresponds to an action taken from the selected node. We will evaluate this node by sampling it. To do so, we go through successors using a default policy until we reach a terminal state, this is usually called a rollout. We then backpropagate the evaluation of that terminal state.

The UCT algorithm is MCTS with UCB as its tree policy.

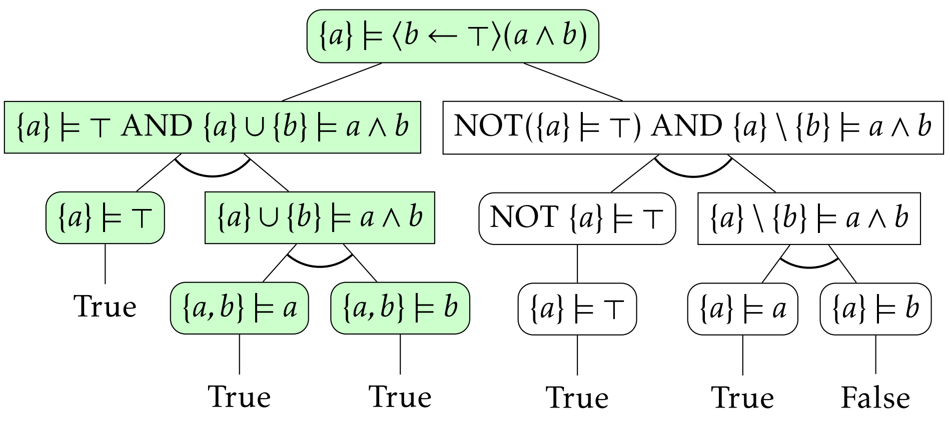

$\{a\} \models \langle b \leftarrow \top \rangle (a \wedge b)$
$\{a\} \models \top$ AND $\{a\} \cup \{b\} \models a \wedge b$
NOT($\{a\} \models \top$) AND $\{a\} \setminus \{b\} \models a \wedge b$
$\{a\} \models \top$
$\{a\} \cup \{b\} \models a \wedge b$
NOT $\{a\} \models \top$
$\{a\} \setminus \{b\} \models a \wedge b$
True
$\{a,b\} \models a$
$\{a,b\} \models b$
$\{a\} \models \top$
$\{a\} \models a$
$\{a\} \models b$
True
True
True
True
False

Figure 2: The tree built from $\{a\} \models \langle b \leftarrow \top \rangle (a \wedge b)$

3 DL-PA as a game

The goal is to design an algorithm to answer the model-checking problem $V \models \varphi$. We build a game equivalent to this problem. This boils down to creating a game tree out of the formula φ.

By doing so, we can directly use the MCTS algorithm. For example, to solve $V \models \psi \vee \phi$ we can either try to prove $V \models \psi$ or $V \models \phi$. It is not clear which formulae will be the easiest to prove. Using Monte Carlo rollouts to evaluate the decisions, the algorithm can take the easiest and fastest way to solve the formula.

3.1 Game tree for DL-PA Model-checking

We want to solve the model checking problem $V \models \varphi$. To do so, each rule of DL-PA's grammar is associated with an intermediate node or a leaf. This defines a tree by induction on DL-PA's grammar. We go through one example by associating $\{a\} \models \langle b \leftarrow \top \rangle (a \wedge b)$ with its tree, as can be seen in Figure 2. AND nodes are rectangles and have an arc between the edge going to the children while OR nodes are rounded rectangles. We colored in green the nodes that constitute the certificate's subtree. This certificate is the proof that $V \models \varphi$ is true.

Deconstruction of a formula

$V \models p$	*if* $p \in V$
$V \models \neg p$	*if* $p \notin V$
$V \models \top$	*if* the world is round
$V \models \bot$	*if* the world is flat
$V \models \bigvee_{i=1}^{n} \varphi_i$	*if* $V \models \varphi_1$ OR $V \models \varphi_2$ OR $\ldots$ $V \models \varphi_n$
$V \models \bigwedge_{i=1}^{n} \varphi_i$	*if* $V \models \varphi_1$ AND $V \models \varphi_2$ AND $\ldots$ $V \models \varphi_n$

Deconstruction of a program inside a modality$\langle \pi \rangle$

$V \models \langle p \leftarrow \psi \rangle \varphi$	*if* $(V \models \psi$ AND $V \cup \{p\} \models \varphi)$ *OR* $(V \not\models \psi$ AND $V \setminus \{p\} \models \varphi)$
$V \models \langle \pi_1 ; \pi_2 \rangle \varphi$	*if* $V \models \langle \pi_1 \rangle (\langle \pi_2 \rangle \varphi)$
$V \models \langle \bigcup_i^n \pi_i \rangle \varphi$	*if* $V \models \langle \pi_1 \rangle \varphi$ OR $V \models \langle \pi_2 \rangle \varphi$ OR $\ldots V \models \langle \pi_n \rangle \varphi$
$V \models \langle \psi ? \rangle \varphi$	*if* $V \models \psi$ AND $V \models \varphi$
$V \models \langle \pi^* \rangle \varphi$	*if* $V \models \langle \pi^{\leq 2^{\lvert \mathbb{P}_\pi \rvert}} \rangle \varphi$
$V \models \langle \pi^{\leq n} \rangle \varphi$	*if* $V \models \langle \pi \rangle \big(\langle \pi^{\leq n-1} \rangle \varphi \big)$ OR $V \models \varphi$

Deconstruction of a box (only the cases that differ from the diamond are shown)

$V \models [\bigcup_i^i \pi_i] \varphi$	*if* $V \models [\pi_1] \varphi$ AND $V \models [\pi_2] \varphi$ AND $\ldots V \models [\pi_n] \varphi$
$V \models [\pi^{\leq n}] \varphi$	*if* $V \models [\pi^n] \varphi$ AND $V \models [\pi^{\leq n-1}] \varphi$

The tree is made so that there is as little as possible overlap between children. Some overlap is unavoidable in nondeterministic cases such as the Kleene star.

$$V \models \langle \pi^* \rangle \varphi \qquad\qquad \textit{if } V \models \langle \pi^{\leq 2^{\lvert \mathbb{P}_\pi \rvert}} \rangle \varphi$$

This deconstruction is justified by the equality $\lVert \pi^* \rVert = \left\lVert \pi^{\leq \lvert 2^{\mathbb{P}_\pi} \rvert} \right\rVert$ in [BHT13], with $\mathbb{P}_\pi$ the set of propositional variables occurring in π. By definition we have $\pi^{\leq n} = \bigcup_{i=0}^n \pi^i$. We can see that it is the nondeterministic union of the empty program with π, and with $\pi ; \pi$, ...

For example we could model a game where the starting state is V; we have n actions π_i and for each action we have a precondition (we cannot do the action if the condition is not met) ψ_i on the state; φ is the winning condition. We could ask whether there exists any sequence of actions that can reach the final state described by a formula φ from the starting state encoded in a valuation V. To do so we can solve the model checking problem $V \models \langle (\bigcup_i^n (\psi_i ? ; \pi_i))^* \rangle \varphi$.

3.2 Size of the tree

We provide a function to measure how many nodes are in the tree corresponding to a DL-PA formula φ. We add $\pi^{\leq n}$ to $\mathcal{L}_{formulae}$ to form $\mathcal{L}'_{formulae}$. This is not included in our DL-PA definition because we only use it to rewrite the Kleene star.

$$m : \mathcal{L}'_{formulae} \rightarrow \mathbb{N}$$

$$
\begin{aligned}
m(p) = m(\neg p) = m(\top) = m(\bot) \quad &= 1 \\
m(\bigvee_{i=1}^{n} \varphi_i) \quad &= 1 + \sum_{i}^{n} m(\varphi_i) \\
m(\bigwedge_{i=1}^{n} \varphi_i) \quad &= 1 + \sum_{i}^{n} m(\varphi_i) \\
m(\langle p \leftarrow \psi \rangle \varphi) \quad &= 2 + 2 * m(\varphi) + 2 * m(\psi) \\
m(\langle \pi_1 ; \pi_2 \rangle \varphi) \quad &= m(\langle \pi_1 \rangle (\langle \pi_2 \rangle \varphi)) \\
m(\bigcup_{i}^{n} \langle \pi_i \rangle \varphi) \quad &= 1 + \sum_{i}^{n} m(\langle \pi_i \rangle \varphi) \\
m(\langle \psi ? \rangle \varphi) \quad &= 1 + m(\psi) + m(\varphi) \\
m(\langle \pi^* \rangle \varphi) \quad &= m(\langle \pi^{\leq 2^{|\mathbb{P}_\pi|}} \rangle \varphi) \\
m(\langle \pi^{\leq n} \rangle \varphi) \quad &= 1 + m(\langle \pi \rangle \left(\langle \pi^{\leq n-1} \rangle \varphi \right)) + m(\varphi)
\end{aligned}
$$

This measure will be used for an algorithm in section 4.

4 Algorithms for the model checking problem

4.1 Monte Carlo Tree Search

4.1.1 MCTS algorithm

In section 3, we have a tree associated with the formula. Thus we can directly use the MCTS algorithm defined in section 2.2.2. We describe here the details of the MCTS algorithm.

For the selection step, we take UCB as the tree policy that selects the most important node at each step to expand. UCB has a parameter C that we have set at its theoretical optimal value $\sqrt{2}$. However in the following section, we will experimentally find a better value for the parameter C.

To evaluate an intermediate node we choose a child at random until we reach a final state that can be evaluated, this process is called a roll-out. This step will be described in more details in the following sections. To evaluate a final state, if the formula is true we give a reward of 1, else a reward of 0.

Updating is done as described. Expanding is done by following the rules to deconstruct DL-PA formulae given in section 3.

This defines the MCTS algorithm for DL-PA formulae.

In the following sections, we introduce transposition tables and this transform the tree into a Directed Acyclic Graph (DAG). It could be beneficial to prioritize nodes in the selection that have many parents. When the node is solved, all the parents will benefit from the result.

4.1.2 Selection & Heuristics for selection

The tree corresponding to a formula has three types of nodes : AND, OR and NOT nodes. Selection of the best child to explore is not the same for OR and AND nodes. NOT nodes have only one child. For an OR node, we select the child that is the most likely to be true, as we only need to establish one of the child to be true to resolve the OR node. For an AND node, we select the child that is the most likely to be false for the same reasons.

We describe here a few heuristics that will help decide whether a node should be prioritized or penalized during the selection process of MCTS.

We can estimate how much time it will take to evaluate an intermediate node. If we have to select one node among others, we can associate a cost of expanding to each node. The cost is be the time taken to perform a complete rollout. For a formula that generates a deep game tree, reaching a final node will take longer. We are using random rollouts and those are usually very fast so the cost is negligible. By considering the cost of evaluating a node we have a multi-armed bandit with costs and it is possible to modify UCB to take them into account [DQZL13].

However we can also estimate how much time the algorithm will take to solve the node. We can prioritize nodes that are smaller and faster to solve completely.

4.1.3 Different rollouts

We first describe how a rollout is done, then provide a possible improvement.

The game tree has three types of nodes : AND, OR, NOT. A choice has to be made with both AND, OR nodes. When faced with an OR node, the rollout process will randomly choose one of the child. With an AND node, the rollout will return the conjunction of the rollouts of each child.

It is possible to only evaluate one child of an AND node, the rollout will be faster but less accurate than the alternative. To represent the inaccuracy, the reward could be lessened. For example if we go through an AND node and we only go through one child out of two, we could give half the reward.

Another possibility we will explore in the experimental section is to do more than one rollout and average the results. This will increase the accuracy but will slow down the algorithm.

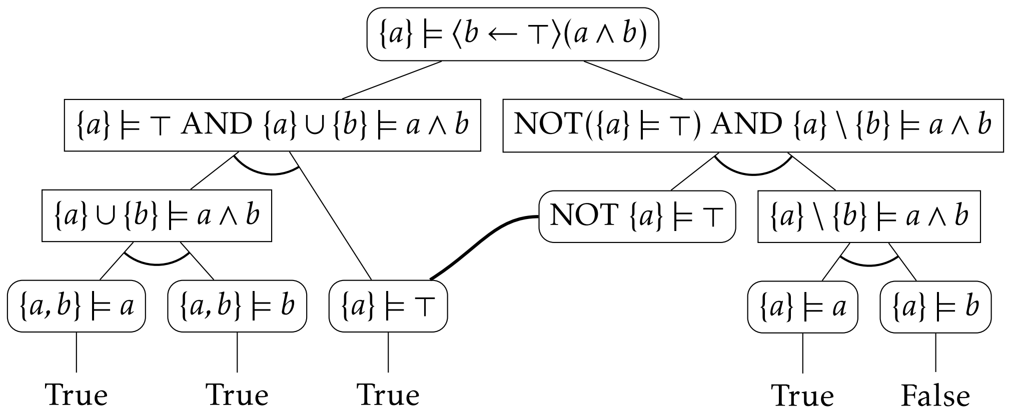
4.2 Transposition table

We modify MCTS to use a transposition table, we will refer to this algorithm as Monte Carlo Dag Search (MCDS). The table is a cache of previously seen nodes. Identical couples of (valuation, formulae) are considered as the same node. A hash function is used. To ensure that we are actually considering logically equivalent formulae, we reduce the valuation by only considering variables that are used in the formula : we register $(\{a,b\},a)$ as $(\{a\},a)$.

The tree is now a DAG as some nodes have multiple parents. The graph may have cycles by evaluating the Kleene star, but we remove them. To do so, when evaluating a star, if we reach a node that we have already evaluated, we remove that node.

For example, to solve a formula that contains an assignment, we can ensure that we do not solve ψ twice by using transposition tables. This can be seen on Figure 3.

After evaluating a node with multiple parents, updating the statistics can be done in several ways. After the selection step, we have traversed the DAG and we remember



$\{a\} \models \langle b \leftarrow \top\rangle(a \wedge b)$
$\{a\} \models \top$ AND $\{a\} \cup \{b\} \models a \wedge b$
NOT($\{a\} \models \top$) AND $\{a\} \setminus \{b\} \models a \wedge b$
$\{a\} \cup \{b\} \models a \wedge b$
NOT $\{a\} \models \top$
$\{a\} \setminus \{b\} \models a \wedge b$
$\{a,b\} \models a$
$\{a,b\} \models b$
$\{a\} \models \top$
$\{a\} \models a$
$\{a\} \models b$
True
True
True
True
False

Figure 3: The use of a transposition table produce a DAG

this path. One naive way to update is to do the same as MCTS by only updating the nodes along the path. This results in some loss of information.

Another way is to update every parent of the node, and every parents of the parents and so on. However by doing this the algorithm is not guaranteed to choose the best action even after many evaluations[SCM12].

4.3 Kleene star deconstruction

$$\begin{aligned} V &\models \langle \pi^* \rangle \varphi && \textit{if } V \models \langle \pi^{\leq 2^{|\mathbb{P}_\pi|}} \rangle \varphi \\ V &\models \langle \pi^{\leq n} \rangle \varphi && \textit{if } V \models \langle \pi \rangle \big(\langle \pi^{\leq n-1} \rangle \varphi \big) \text{ OR } V \models \varphi \end{aligned}$$

Deconstructing the Kleene star this way makes it clear that the tree is finite, but the most important point is that it solves a problem we would encounter if we deconstructed the Kleene star as an union $\pi^{\leq n} = \bigcup_{i=0}^{n} \pi^i$. The problem is that this way generates an OR node with n children. When MCTS is faced with such a node, it will first explore once every children. n is in practice very large, with 20 variables used in π, $n = 2^2 0 = 1048576$. Even simply storing every children in working memory could become a problem, but the algorithm would take a lot of time without progressing towards a solution. Thus deconstructing by dividing the Kleene star into two problems is better adapted to the algorithm at hand.

4.4 Assignment deconstruction

We recall that in the base algorithm, the assignment is deconstructed in the following way.

$$\begin{aligned} V \models \langle p \leftarrow \psi \rangle \varphi \qquad & \textit{if } \ (V \models \psi \text{ AND } V \setminus \{p\} \models \varphi)\, OR \\ & (V \not\models \psi \text{ AND } V \cup \{p\} \models \varphi) \end{aligned}$$

One drawback is that in MCTS, the formula ψ can be explored twice. This is solved by using MCDS. Exploration has to be careful, as it is possible that ψ is simple to solve and true, which means that all the time spent on the branch that consider ψ false is wasted.

The algorithm that uses Oracles deals with assignment by evaluating if ψ is small, and if it is, it calls the oracle to get the result of ψ and the assignment is then only one node.

The assignment can be done in another way :

$$\begin{array}{ll}
V \models \langle p \leftarrow \top \rangle \varphi & \textit{if } V \cup \{p\} \models \varphi \\
V \models \langle p \leftarrow \bot \rangle \varphi & \textit{if } V \setminus \{p\} \models \varphi \\
V \models \langle p \leftarrow (\psi_1 \vee \psi_2) \rangle \varphi & \textit{if } V \models \langle p \leftarrow \psi_1 \rangle \varphi \\
 & \quad \text{OR } V \models \langle p \leftarrow \psi_2 \rangle \varphi \\
V \models \langle p \leftarrow \langle \pi_1 \cup \pi_2 \rangle \psi \rangle \varphi & \textit{if } V \models \langle p \leftarrow \langle \pi_1 \rangle \psi \rangle \varphi \\
 & \quad \text{OR } V \models \langle p \leftarrow \langle \pi_2 \rangle \psi \rangle \varphi
\end{array}$$

We deconstruct based on what is inside the assignment's formula. While this decreases the size of the formula inside, we increase the total formula size by duplicating φ. Similarly, we have to carefully stop the exploration of the other branch once one of ψ_1, ψ_2 has been found to be true. This deconstruction of the assignment does not fit well with the AND/OR tree.

4.5 Solving without Monte Carlo

As we have defined a game tree, the simplest solver is to use a depth first traversal of the game tree. However an unguided search for the Kleene star leads to poor performances as it has to solve many duplicates.

We design a simple solver based on the depth first algorithm, we will refer to it as the simple solver. The simple solver differs from a depth first traversal by treating the Kleene star differently.

Algorithm 1 Solve $V \models \langle \pi * \rangle \varphi$

```
$ToCheck \leftarrow \{V\}$                  ▷ $ToCheck$ and $Checked$ are sets of valuations
$Checked \leftarrow \emptyset$
while $ToCheck \neq \emptyset$ do
    $Next \leftarrow Pop(ToCheck)$         ▷ Pop returns one element of $ToCheck$
    if $check(Next, \varphi)$ then
        Return $true$
    end if
    $Checked \leftarrow Checked \cup \{Next\}$
    $ToCheck \leftarrow (apply(next, \pi) \cup ToCheck) \setminus Checked$
end while
Return $false$
```

$apply(V, \pi)$ is the function that applies the program π to the valuation V. For example : $apply(\{a\}, b \leftarrow a\top) = \{\{a, b\}\}$. Recall that a program can be non deterministic : $apply(\{a\}, b \leftarrow \top \cup b \leftarrow \bot) = \{\{a, b\}, \{a\}\}$ so the function apply returns a set of valuations. $check(V, \varphi)$ the function that return the result of $V \models \varphi$, with V a valuation, φ a formula.

Informally the program applies π once and stores all the exit valuations (the valuations reached after applying the program) in a queue. It then picks a valuation out of the queue, taking care to never select the same valuation twice by storing the set of all the valuation it has already applied π, and has already verified it does not make φ true. This goes on as long as the queue is not empty. When it is empty, we know that applying π again will only lead to already verified valuations. As we've mentioned earlier, the program π contains a finite number of variables and thus can only produce a finite amount of different valuations. Therefore this algorithm terminates.

As we will seen in the experimental section, remembering the valuation we have already treated is key to solving those model-checking problem in reasonable time.

4.6 Using simpler solvers for simple formulae

In this section we describe a new algorithm based on MCTS. The goal is to reduce the amount of nodes and let faster algorithms (now called oracle) take care of the simple problems, leaving the time for the more complex algorithm to take more impactful decisions.

The algorithm has to evaluate an intermediate node. Heuristics will decide whether to call for an oracle, or to expand the node and continue the search with more precision. The oracle can be any of the solvers we described.

The heuristics can decide whether to call the oracle based on multiple factors listed below. All of them are implemented and compared experimentally in section 5

- if the formula is union and Kleene star free (deterministic)
- if the formula is Kleene star free
- if the formula is propositional
- if the formula is small (measure m described in section 3

We present another but unimplemented idea on how to use simpler solvers. Every new node would be solved by the Oracle, but with a timeout. For example it would immediately solve every formula that takes less than a second, and if it didn't we know it would be worth creating a new node to simplify the problem.

5 Experimental comparison of algorithms

5.1 Experiments

I have implemented all algorithms in OCaml. The code is mono-core and compiled with O3 optimisations. The code is available at https://github.com/vmerckle/mctsdlpag. The experiments are orchestrated on multiple cores by a python script that uses a worker pool which can be found on the same git. I have ran all the experiments on a ryzen 5 2600 at 3.9Ghz on Debian. The first computer I ran the experiment on has experienced thermal throttling, the performance would drop over time as the processor would run slower to lower the heat. I made sure this did not happen on the next computer by controlling the frequency at which the cores ran.

Execution time is measured by elapsed cpu time. We take multiple samples, at least 10 for each number presented below. The execution time takes into account the whole program execution, from initialization to printing the result. This way of measuring does introduce inaccuracy. However these inaccuracies fade as we solve instances that takes up to a few minutes to solve, and we measure each algorithm the same way. A timeout of 10s indicate that we terminate the solver after 10 seconds. For example, the problem Counter5000 (to count in unary to 5000) takes 5 seconds to parse and process before starting the solver.

The first step was to verify if the algorithm correctly solve formula and has no bugs. We ran the algorithms on simple test-cases that could be solved by hand, they consisted of all the primitive of the DL-PA language and then more complex examples that had obvious solutions.

Then the algorithms were slightly optimized. I rewrote both the MCTS (stores the nodes in a tree structure) and MCDS (using a hash table to store the nodes) to reduce the hash table usage, and optimize the rollout function that are heavily used. (On the github, MCTS and MCDSv2 are the rewritten versions, MCDS is the first version.)

5.2 Domains

We describe the different model-checking problems we use to compare our algorithms. We also show how the encodings are done.

Counter(n) is the unary counter described in the background section(2).

$$\text{Count} = \bigcup_{i=1}^{n} (Val(i-1)?\,;Val(i) \leftarrow \top\,;Val(i-1) \leftarrow \bot)$$
$$\{Val(0)\} \models \langle \text{Count}^* \rangle Val(n)$$

Counter(n) has n variables, and to succeed we have to apply the program under the Kleene star n times in a row. Counter has on average a high branching factor, as each step of the counter is a union of n programs.

Hanoï(n,m) is the time taken to solve the Hanoï puzzle with n stacks and m different pieces (the usual puzzle for humans has 3 stacks and 7 pieces). The puzzle starts with all the pieces on first stack. To solve the puzzle, we have to move every pieces from the first stack to the last stack. This is a more complex puzzle. There are $n * m$ variables. To encode the Hanoï puzzle in DL-PA, we describe in a program $play$ all the possible moves with their precondition. For example, for the action a that moves the bottom piece from stack 1 to stack 3, we have to check ψ_a if it is currently on the stack 1 and if there is no pieces above it. This move would be encoded as $\psi_a?;a$. Then we describe the final state in a formula φ, this is simply the logical description of the final state. The play program with two actions a, b with two preconditions ψ_a, ψ_b would look like this : $play = (\psi_a?;a) \cup (\psi_b?;b)$. The model checking problem is then $\{initial\_state\} \models \langle play^* \rangle \varphi$. This encodes the decision problem : is there a way to solve the Hanoï puzzle?

The problem "Hanoï(3,4)ez" is the Hanoï(3,4) problem except we moved the second from bottom piece to the second stack. This is closer to the solution and provides a problem of intermediate difficulty for the solvers.

Table 1: timeout=100, samples=100, Execution time in seconds

Problem	Simple	Naive
Hanoi(3,1)	0.010	0.010
Hanoi(3,2)	0.011	0.010
Hanoi(3,3)	0.011	0.031
Hanoi(3,4) ez	0.016	2.8
Hanoi(3,4)	0.014	8.8
Hanoi(3,5)	0.019	SO
Counter(1000)	0.20	SO

Table 2: timeout=100, samples=100, Execution time in seconds

Problem	MCTS	MCDS	MCTS nodes	MCDS nodes
Hanoi(3,1)	0.010	0.010	113	39
Hanoi(3,2)	0.019	0.013	1122	585
Hanoi(3,3)	3.2	2.3	226175	83263
Hanoi(3,4) ez	TO	TO		
Hanoi(3,4)	TO	TO		
Counter(1000)	TO	TO		
Counter(10000)	TO	TO		

5.3 Adjusting the parameters of each algorithms

In this section, timeout is the amount of time before an algorithm is forced to terminate and is marked "TO" in the table. If the algorithm runs out of stack, we write "SO".

5.3.1 Best solver without Monte Carlo

First we evaluate the performance of the two solvers that do not use Monte Carlo. We refer to the depth first algorithm by "Naive" in table 1.

The depth first algorithm quickly reaches its limit by having Stack Overflow errors (SO in the table). Kleene stars can produce very deep trees, and the Naive algorithm is implemented recursively. The simple solver gets around the stack limit by treating Kleene stars in an imperative way, and by never verifying the same valuation twice as described in section 4. The simple solver is also faster than Naive on bigger instances of Hanoï.

We will use the simple solver as the oracle in the rest of the paper.

5.3.2 Using a transposition table : MCTS vs MCDS

We test the base MCTS algorithm described in section 2 and 4, against the MCTS with transposition table from section 4 that we will call MCDS.

In table 2 we can see that the algorithm with transposition tables (MCDS) beats the normal algorithm. MCDS only needs to evaluate half the number of nodes of MCTS for the Hanoï puzzle.

We do not exclude MCTS as of yet, as the results are still similar.

Table 3: timeout=100, samples=100, Execution time in seconds

Problem	MCDS	MCDS 1	MCDS 2	MCDS 3	MCDS 4
Hanoi(3,1)	0.013	0.0094	0.0094	0.0094	0.0096
Hanoi(3,2)	0.013	0.011	0.012	0.011	0.019
Hanoi(3,3)	2.4	0.56	0.60	0.62	1.9
Hanoi(3,4) ez	TO	153.0	162.0	155.0	251.0
Hanoi(3,4)	TO	TO	TO	TO	TO
Counter(180)	92.0	92.0	90.0	93.0	133.0
Counter(500)	TO	TO	TO	TO	TO

5.3.3 Best oracle condition

We described in section 4 a new algorithm, which modifies the expansion step of MCTS. Instead of always creating a new node when we encounter a formula, we use a function (here called condition) that decides whether to create a new node, or to call an oracle that will directly solve the formula. The oracle used here will be the simple solver.

We tested four different conditions listed below :

- 1 : if the formula is propositional
- 3 : if the formula is union and Kleene star free (deterministic)
- 2 : if the formula is Kleene star free
- 4 : if the formula is small, will produce a small amount of nodes

We test them against the base algorithm (MCDS), which always create a new node. For example, MCDS 1 refers to the new algorithm that uses transposition table and the decider function 1: it directly solve the formula if the formula is propositional.

We can see in table 3 that using oracles is an improvement for the Hanoï puzzle, and that the propositional oracle is the most efficient. The counter180 problem does not have many small formulas and mostly consist of a large Kleene star. From the results on the counter problem, we can see that evaluating the size of the formula is too costly to be used, and that the cost of the decider is not negligible. One improvement could be to store the size of each formula at startup time.

The results are similar for MCTS : the best condition is to use the oracle on propositional formulae. In table 4 we do a comparison between transposition (MCDS) and without (MCTS) with the use of oracle on all propositional formulae. MCDS still has the upper hand with the addition of oracles.

5.3.4 UCB's constant variation

UCB refers to the MCTS algorithm that uses UCB as its tree policy. All our Monte Carlo algorithms uses the formula below to select the next node to expand :

$$U(i) = \frac{r_i}{n_i} + C\sqrt{\frac{\ln n}{n_i}}$$

Table 4: timeout=100, samples=50, Execution time in seconds

Problem	MCTS 1	MCDS 1
Hanoi(3,1)	0.011	0.0098
Hanoi(3,2)	0.015	0.012
Hanoi(3,3)	1.8	0.57
Hanoi(3,4) ez	TO	154.0
Counter(180)	114.0	101.0

Table 5: Hanoi(3,3), timeout=3, samples=100,Execution time in seconds

C constant	MCTS propositional	MCDS propositional
0.0001	TO	0.27
0.001	TO	0.28
0.01	TO	0.27
0.03	0.48	0.25
0.07	0.98	0.26
0.09	0.96	0.26
0.1	0.93	0.27
0.4	1.5	0.35
0.8	1.7	0.51
1	1.7	0.55
1.2	1.9	0.58
1.4	1.9	0.59
1.6	1.8	0.62
1.8	2.1	0.62
2	2.2	0.63
4	2.5	0.69
8	2.7	0.74
1024	TO	0.80

Table 6: timeout=100, samples=20,Execution time in seconds

Nb rollout	MCDS 1	MCDSn 1	MCDSn
1	0.58	0.57	2.4
2	0.54	0.59	2.3
10	0.53	0.68	2.8
20	0.52	0.80	3.3
100	0.53	1.6	6.6

Table 7: timeout=150, samples=10,Execution time in seconds

Nb rollout	MCDS 1	MCDSn 1	MCDSn
1	92.0	90.2	92.0
2	89.0	91.0	89.0
10	88.0	91.0	89.0
20	91.0	90.7	92.0
100	88.0	93.0	88.0

The UCB formula is composed of two terms that allow for a balance between exploitation and exploration. The first term r_i/n_i is the exploitation term, and the second with the parameter C attached is the exploration term. The parameter C is equal to $\sqrt{2}$ in theory. However as we can see in table 5 there is room for improvement. If UCB has a smaller C value, it will prioritize exploitation. If the rollouts are accurate, the intermediate evaluations of nodes are accurate as well. The algorithm can thus focus on the best nodes. If we completely neglect exploration, the nodes that had bad luck will get explored last and this will slow down the algorithm on average.

From the results of table 5 we can see that reducing C will speed up the algorithm up to a point. This can be explained by the fact that rewards are very rare in some encodings. This means that the exploration terms is quickly dominant in the selection process. Decreasing C will allow the algorithm to follow the trail that leads to the rare rewards.

5.3.5 Increasing the number of rollouts per turn

We experiment the effect of one modification to the expansion step. Usually one rollout is done to evaluate the new node. In table 6 we test the performance of MCDS when we use more than one rollout and average the results on the Hanoï puzzle. Performance is decreasing because the new playout are not accurate enough to keep up with the additional cost of creating new playouts. In table 7 we run the same experiment but on the counter encoding, the differences are negligible.

Since the rollout accuracy changed, we need to optimize the C constant again. As we see on table 8, the best parameter differs depending on the number of rollouts.

Table 8: samples=100, Execution time in seconds

C	n=1	n=2	n=4	n=10	n=20	n=100
0.0001	0.27	0.25	0.25	0.25	0.30	0.72
0.001	0.27	0.26	0.25	0.26	0.30	0.72
0.01	0.26	0.26	0.24	0.25	0.31	0.72
0.03	0.27	0.26	0.24	0.26	0.31	0.72
0.07	0.26	0.25	0.25	0.26	0.31	0.72
0.09	0.27	0.27	0.24	0.27	0.31	0.72
0.1	0.27	0.26	0.26	0.27	0.32	0.72
0.2	0.27	0.26	0.26	0.29	0.34	0.80
0.4	0.34	0.35	0.37	0.44	0.52	1.1
0.6	0.47	0.47	0.48	0.53	0.63	1.2
0.8	0.50	0.53	0.57	0.63	0.75	1.5
1	0.55	0.56	0.60	0.67	0.80	1.6
1.5	0.60	0.62	0.64	0.73	0.86	1.7
2	0.62	0.64	0.67	0.76	0.88	1.7
4	0.68	0.69	0.72	0.82	0.94	1.8
8	0.74	0.75	0.80	0.88	1.03	1.9
1024	0.81	0.82	0.85	0.95	1.08	2.03

5.4 Comparison of the best parameters

Finally we compare MCTS, MCDS, MCDSn2 all using the oracle on propositional formulae and with $C = 0.0001$ and the simple solver.

In table 9, we can see the simpler solver is very effective on straightforward instances. However we included the problem "simpletrap" which is simply this formula : $\{a\} \models hard \lor easy$. The easy problem is simply $a \land a \land a \land a$, thus any rollout made on the easy branch will return a positive reward. The hard problem is the opposite, any rollout will return a negative reward. The hard problem chosen here is to count to 5000, and ask if we counted to 5000 which is always false. We can see that this way, we can construct an example where the simple solver is worse than Monte Carlo methods.

By using oracles, we tried to bridge the gap between the simple solver and Monte Carlo solvers. However analysing the formula can be costly as it is not obvious whether a Kleene star can be directly solved.

Further work & Conclusion

We have designed the first algorithms to solve DL-PA formulae and we tested them on different DL-PA problems. This required the development of a DL-PA parser, the solvers themselves and scripts to plan the experiments. The Monte Carlo solvers were able to solve small Hanoï instances, but the simple solver is more efficient and could solve much bigger instances. However it is shown that the Monte Carlo algorithms are promising for more complex instances as they do look ahead and focus on the most likely candidates.

There are still a lot of ideas left to explore to solve DL-PA problems. It has been shown that solvers can have drastically different performances on slitghtly similar en-

Table 9: timeout=25200, samples=100, Execution time in seconds

Problem	MCTS 1	MCDS 1	MCDSn2 1	Simple
Simple trap	TO	4.2	3.7	8.8
Hanoi(3,1)	0.010	0.0099	0.010	0.0099
Hanoi(3,2)	0.015	0.011	0.011	0.010
Hanoi(3,3)	0.93	0.26	0.26	0.011
Hanoi(3,4) ez	TO	153.0	150.0	0.012
Hanoi(3,4)	TO	TO	TO	0.013
Hanoi(3,5)	TO	TO	TO	0.014
Hanoi(3,6)	TO	TO	TO	0.025
Hanoi(3,7)	TO	TO	TO	0.061
Hanoi(3,8)	TO	TO	TO	0.19
Hanoi(3,9)	TO	TO	TO	0.78
Hanoi(3,10)	TO	TO	TO	2.4
Hanoi(3,11)	TO	TO	TO	7.4
Hanoi(3,12)	TO	TO	TO	25.0
Hanoi(3,13)	TO	TO	TO	86.0
Hanoi(3,14)	TO	TO	TO	280.0
Hanoi(3,15)	TO	TO	TO	930.0
Hanoi(3,16)	TO	TO	TO	4700.0
Counter(180)	140.0	120.0	110.0	0.015
Counter(500)	TO	TO	TO	0.055

codings. It is known to be difficult to estimate how hard a SAT formula will be, but there is some hope that DL-PA encodings are more easier to estimate. Going forward, the next algorithm should estimate how likely a formula is to be true and at the same time, estimate how much effort it will be to find out its truth value. When those two estimates are combined, smarter decisions can be made. This will at least avoid the traps like the one designed for our simple solver. It is also possible to use neural networks to either take the decisions, or to do the estimates as it has been tried for SAT [XL21].

The best SAT solvers use conflict-driven clause learning. We made efforts to find an equivalent method for DL-PA. It is hard to apply it to DL-PA because what we learn at a conflict is only locally useful because of assignment. We explored the idea of another deconstruction of a formula that would lead to a less deeper tree, by adding variables and composing the programs as much as possible inside modalities. This way the assignment would be toward the top of the tree. The results were too sketchy to include in this report.

Experiments were important at each step, however as the number of parameters and the difficulty of problems increased, it became harder to conclude. When an experiment is taking longer than expected, it would be useful to know how long it would require to finish. I started to design a way to evaluate the intermediate state of a MCTS, by applying a threshold to each decision. If the algorithm is 90% sure of the right node, we would cut the part of the tree that the algorithm has found to be unlikely to be true. We could compute the percentage of the tree we have removed yet. This measure could also lead to a new algorithm.

References

[ACF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256, 2002.

[BHST14] Philippe Balbiani, Andreas Herzig, François Schwarzentruber, and Nicolas Troquard. DL-PA and DCL-PC: model checking and satisfiability problem are indeed in PSPACE. *CoRR*, abs/1411.7825, 2014.

[BHT13] Philippe Balbiani, Andreas Herzig, and Nicolas Troquard. Dynamic logic of propositional assignments: A well-behaved variant of PDL. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 143–152. IEEE Computer Society, 2013.

[BPW+12] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intell. AI Games*, 4(1):1–43, 2012.

[DQZL13] Wenkui Ding, Tao Qin, Xu-Dong Zhang, and Tie-Yan Liu. Multi-armed bandit with budget constraint and variable costs. In Marie desJardins and Michael L. Littman, editors, *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*. AAAI Press, 2013.

[SCM12] Abdallah Saffidine, Tristan Cazenave, and Jean Méhat. UCD : Upper confidence bound for rooted directed acyclic graphs. *Knowl. Based Syst.*, 34:26–33, 2012.

[SS96] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In Rob A. Rutenbar and Ralph H. J. M. Otten, editors, *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*, pages 220–227. IEEE Computer Society / ACM, 1996.

[XL21] Ruiyang Xu and Karl J. Lieberherr. Solving QSAT problems with neural MCTS. *CoRR*, abs/2101.06619, 2021.

A Contexte

J'ai effectué mon stage de 5 mois dans l'équipe LILaC hébergée à l'IRIT à Toulouse. Malgré la pandémie et les confinements, j'ai pu alterner entre travailler au laboratoire et à la maison et profiter d'une bonne ambiance. Je remercie mes trois superviseurs qui ont su me guider durant ce stage et assister régulièrement à des visioconférences. Je remercie particulièrement Andreas Herzig pour sa grande hospitalité, les bons repas et les balades jusqu'à l'IRIT à vélo ! Ainsi que Abdallah Saffidine qui a toujours eu des remarques et conseils avisés, et a amené de la bonne ambiance à travers les ondes, depuis l'Australie.